Preprint
UCRL-JC-145047

A Visual Language for Data Mapping

J. W. Carlson

This article was submitted to Object Oriented Programming Systems Languages and Applications 2001 Workshop on Domain-Specific Visual Languages, Tampa Bay, FL, October 14-18, 2001

August 10, 2001

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

A proposed position paper submission to the OOPSLA 2001 Workshop on Domain Specific Visual Languages

A Visual Language for Data Mapping

John Carlson
Lawrence Livermore National Laboratory
Carlson14@llnl.gov

Abstract

I discuss a visual programming language and environment for doing data mapping between textual data files. Our experience using the environment is documented, as well as many of the constructs we used in the environment. I relate problems and solutions, and pose some questions to the readers of this document and the workshop.

Introduction

At this workshop and in this document, I choose to report on

- our experience of creating a domain-specific visual language (DSVL) for doing data mapping, sometimes called data translation
- the constructs we identified for doing data mapping
- an example of the constructs
- relationships between constructs
- the code generation process we use
- and how our DSVL hasn't evolved during use, but how it might evolve in the future
- how one adds new features to our DSVL
- what our DSVL isn't
- my questions on meta-"fill in the blank here"
- other opinions

Our experience

We have developed the *Translator's Workbench* and *Engine (TWB/TE)* for use with a repository of "programs" to do data mapping. I use the quotes because the program isn't a conventional textual program, instead it is a program stored as records in files, one per class of object. The TE is an interpreter of those files (or a virtual machine). The TWB is an Integrated Development Environment (IDE) that makes use of the TE library and also adds a graphical user interface which allows software visualization, programming by example, visual programming, and debugging in our domain. The TWB/TE's purpose is to serve as a platform for creating procedural maps from multiple input files to multiple output files, but is being extended to incorporate SQL database commands, commands that provide textual input and output (shell or console commands), and network services. The TWB/TE is imperative--you tell it what you want it to do within the context of input examples you provide. Other mappers that the author is aware of are declarative--you tell the mapper what the format of the files are, and the mapping between individual elements, then it does the mapping in whatever order it chooses.

We created several maps using the TWB/TE. The first map that we created was a demonstration map to convert a file in military format to an Electronic Data Interchange (EDI/X12) file format. We used a

specially formatted file provided by a domain expert and generated the code necessary to create a map, then compiled and ran the code to generate the map repository. Once we had the repository in place, we used the TWB to demonstrate the visual mapping capability. We had the same domain expert use the TWB tool to create more maps that we used in building e-commerce sites at LLNL, Wright-Patterson Air Force Base and the Veterans Administration's Long Beach Medical Center. He wrote several mappings between EDI/X12 and proprietary formats generated and understood by our database routines. On another project he developed mappings to generate reports from X12 invoice data. He also used TWB on the project in which the first secure financial EDI payment over the Internet was issued between LLNL and the Bank of America; he mapped a database table dump of invoice payments to the EDI/X12 820 transaction set.

The TWB is a reversible debugging interpreter. You can set breakpoints, run the mapping backwards and forwards, and you can insert new operations at the current operation—you can actually add new operations while a mapping is running in the TWB, if you desire.

The TE can be used in a production environment to accomplish the mapping task on sets of data in a network environment like EDI. You specify the files you want to map from and to on the command line, or by default it uses standard input and standard output (cin & cout)

Identifying Constructs

We have been adding constructs incrementally as the project proceeds. Some of the first constructs that appeared were the calculator, the table, and the text document. We knew from EDI/X12 that we needed a way to count the number of segments that we were producing, so we created a RPN calculator and the arithmetic operation representing button presses on the calculator. We also knew that we needed to provide a mapping from one set of data to another set, so we created some table classes with a variable number of keys in each row and their corresponding values for each instance. Since EDI/X12 data is packaged in an interchange, we knew we needed a way to read the data in, so we created the receive operation, and the corresponding send operation. We decided to implement these operations as working on files, rather than on the network, and we called these items input documents and output documents, although in reality, they are just plain ASCII text, with some support for binary data added later.

The goal was to provide a "programming" environment, so we needed a way to represent the mapping being done. We chose the recorder which sort of mimics how a glorified tape recorder might work. A couple of people on the project had discovered some papers on reversible interpreters (I don't know the references, they were never given to me), so we decided to make all of the operations undoable. We needed a way to save this data on an undo stack, so the corresponding undo record and undo operations were written. We needed a way to take data from an input document and place the data in either an output document or the calculator, so we created the copy operation. While not necessary for mapping, we wanted a way to hide a document, table, or calculator, so we added a hide operation, and also a show operation when a document, table or calculator is first shown (created within TWB—not necessarily the first place it is used in the TE).

Next, we needed to really focus on how we were going to enable actual "programming", instead of just providing a macro recording capability, so we subclassed the table object to create the branch or conditional. The keys became predicate tests such as "Is filled," "Equal to," "In set," "Anything" and more, mostly determined by the requirements for the first demonstration mapping, and the single value became procedure which could hold several operations. We had decided to display each procedure as a branch of code splitting off from the main branch, alternate branches are shown in parallel to each other (see figure 1). The branch compares the keys copied into the branch against each row of predicates (each vertical branch in the recorder), until a match is found. If no match is found, none of the corresponding procedures are performed, and the program continues after the branch. The branch provides an ordered comparison of groups of and'ed predicates. Since we didn't want to run the procedure before we had copied all the test keys into the branch, we added a Lookup operation to the branch. We also decided that we would apply the Lookup operation within a branch to allow recursion (used for repetitive operations).

The demonstration mapping also requires dates and date operations. The date calculator shows a date in various formats, and provides some operations on dates. The operations on dates are implemented with the copy operation—when a value is copied into a text field on the date calculator, an operation is performed based on which field is copied into.

We implemented several more items before deployment. We added the string calculator and operations which were used in the string calculator, such as trim, truncate, justify and comparison; the move text location operation, which allowed us to do copies into documents relative to a cursor, instead of absolute positioning; the search operation on documents; and the reset operation, which undoes the effect of the move text location operation and puts the cursor at the top of the document. The reset operation was mainly necessary to prepare for the next set of documents to map. We recorded the reset operations and the send operations within the final procedure in the recorder. I had tried to implement an EDI Acknowledgement using the operations previously described, and some additional objects, but gave up, and wrote an Acknowledgement operation instead. This is a very specialized operation that takes EDI in an input document and produces an acknowledgement in an output document. I wrote a shell script that generated the acknowledgement data structures from the EDI/X12 standard description file. I compiled and ran this program to generate a map repository for doing the acknowledgements.

Something we always intended to implement, but was delayed was the form. I finally got the gumption up to do it when I decided to put an XML parser into the mix (I really want to write a parser by example, but haven't gotten to it yet). I present the XML (or EDI/X12) document in outline format in a window. The user can traverse the document using the arrow keys and copy the current type, name, value, and status of the current item in the form. The status is used to determine whether there are any more items to search for using the previously selected arrow key—The value of status is either "OK" or "End."

There are a couple of items that the TWB/TE was used for in the past, namely, writing shell scripts or batch files, and sending commands to a SQL interpreter. We currently have these features working now in the TWB, but the TE doesn't support scripts or batch files yet, because we don't say "get as much data you can until you block", so the TE exits before the script starts producing output. The extend operation is intended to be handle cases like this. The shell script window in TWB has also been used to implement TCP/IP servers, but suffers from similar problems as the scripts. I haven't implemented reversibility for these items yet.

Currently, the all operations appear in one flowchart (the recorder). It would be highly desirable to add separate panels or windows for individual procedures, because the Workbench is very slow when opening and closing deeply nested branches.

The things that the operations are working on (the desktop objects) are shown in separate windows. For example, each calculator, document, branch, and SQL statement is shown in a separate window. Many times, we don't bother to bring up a window unless the user explicitly requests it, or the user is stepping through the mapping and moves across a show operation.

The environment we wrote supports both cross domain and domain-specific programming of maps. Each desktop object can serve as a domain, and new domain handlers can be added. There are operations which cross domains such as the copy operation and operations within a single domain such as the arithmetic operation. The recorder serves as the integration or glue between the domains. The core GUI components such as single and multiline text fields are shared across domains to enable cross domain copying. Next we show an example of the constructs described above

Example of constructs

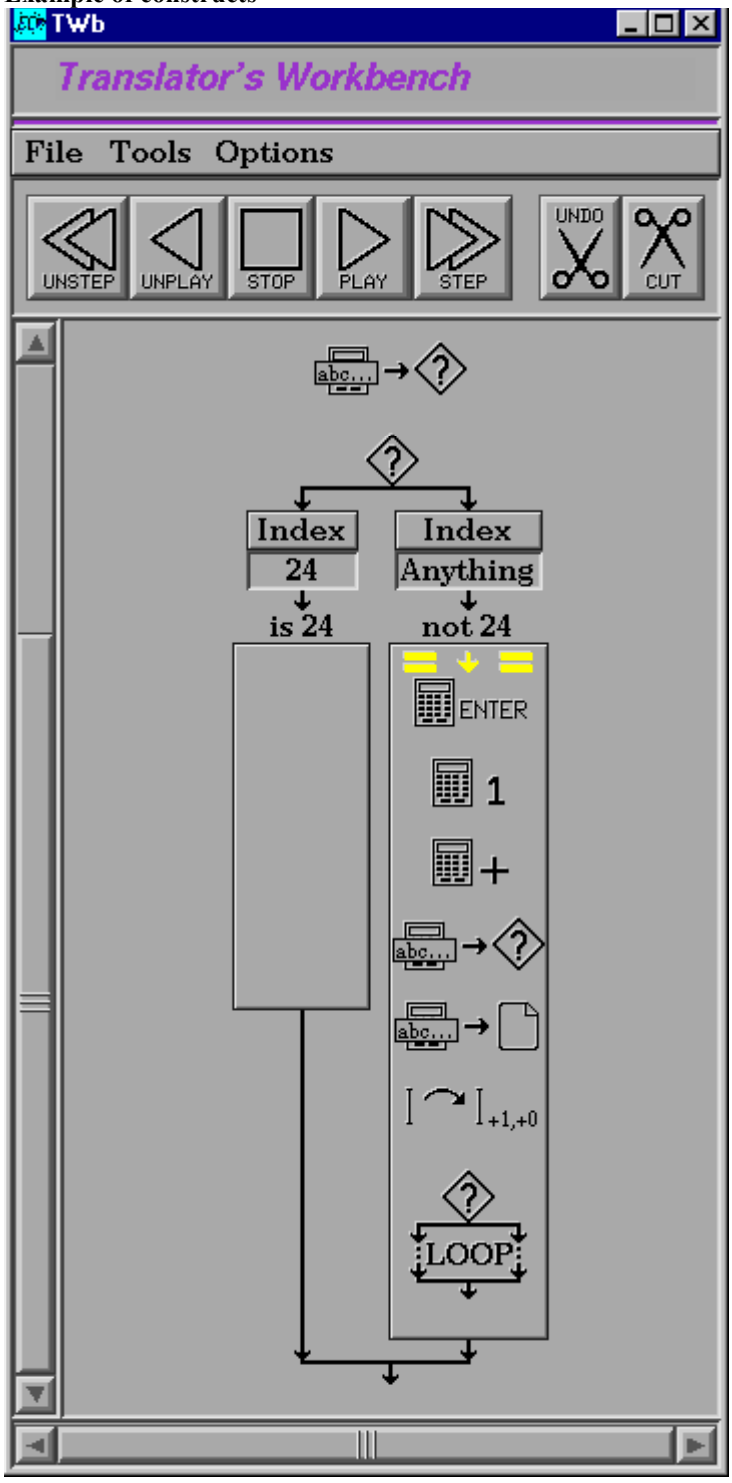

Figure 1. Recorder

Here in figure 1 is an example of the visual language mapping required to create an output document containing a list of numbers, one on each line. Scrolled off the top are several show operations to create new desktop objects, namely, a calculator, a branch and an output document. The first operation that appears is a copy operation that copies from the calculator into the branch, then a compound or lookup

operation appears which is the conditional test in the branch. There are two possible outcomes. Either the item copied into the branch is "24" or it's anything else. The calculator starts at 0. When the value isn't 24, 1 is added to it in the calculator, and the value is copied out of the calculator and into the branch again, and as well, the value is copied into the output document. The cursor is moved to the next line in the output document, and the conditional branch is tested again. When the value is 24, the mapping proceeds down the empty branch, and ultimately, the mapping finishes. We have a menu item in the branch object that does the predicate test. When the predicate test is on a branch that is in an outer branch that contains the same predicate test, an icon is used instead of creating a recursive copy of the same operations (for performance reasons and integrity). The program counter or current operation icon is moved up to the top of the enclosing branch, so that the operations repeat over and over again. However, we keep track of all the branches behind the scenes, so that operations can continue after a branch (or procedure call) finishes. Probably the most complex thing I implemented using this technique with branches has been a depth-first tree traversal of an XML document. We can also do simpler stuff like retrieving rows from a database query.

Relationships between constructs

Below I list the operations supported within the environment. I specify the operation that is recorded, the desktop objects or components that it applies to, and a short description of the operation.

Operation	Applies To	Function
Create/Show	Any Desktop Object	Displays Desktop Object on screen
Hide	Any Desktop Object	Removes Desktop Object from screen
Arithmetic	Calculator	Any button pressed in calculator
Send	Document and Form	Send contents of Document or Form to file
Receive	Document and Form	Receives contents into Document or Form from file
Lookup	Tables except for Data Table	Begins predicate testing, launches guarded procedures
Procedure		A list of operations
Move Text Location	Text Location within Document or Form	Moves Text Location by offset
Search	Text Location within Document	Moves Text Location to a successful search
Reset	Text Location within Document	Undoes effects of Move Text Locations and Searches
Justify	String Calculator	Justifies a string center, left or right, adds padding
Trim	String Calculator	Trims whitespace from a string

Truncate	String Calculator	Truncates characters from a string
Copy	Data Source (Select, Search or Bounded) and Text Location	Copies text in source and overwrites text following Text Location
Ack	Input Document & Output Document	Creates an EDI Acknowledgement
Extend	Form, Command, and SQL	Add a new child (Form), get more output (Command), get next row/execute (SQL)

Table 1. Persistent Operations in TWB/TE

Code Generation

To produce a compiled map, we take the objects from the map, and write them out in C++ as global variables (I ripped off this idea from Tim Sullivan's work). Each object is responsible for writing a constructor call that can be used in this manner. The infrastructure takes care of writing out the entire set of C++ objects. The result is compiled (if you don't run into various compiler bugs!), and linked with the Engine's main() and supporting libraries. This allows us to take the compiled program and run it with a parameter which tells the compiled program to regenerate the repository in a directory that the compiled program is running from. The C++ variables also provide a convenient distribution method. A feature allows the Engine and Workbench to read in the C++ variables, instead of working from the repository or the compiled code (also ripped off the idea). We also wrote a program that allowed someone to browse the C++ variables over the world wide web—The web server would submit the class and object identifier to a CGI program which manipulates the C++ variables (the object and its attributes) into a hyper-linked page.

Code generation may not be the way to go. You may want to create an interpreter for a visual language based in your repository, not an external programming language. Our tool can execute code in a mapping that has never been in a conventional programming language. There is no need for code generation in our tool because it stores both programs and domain-specific objects in its repository. Our repository is a portable, machine-readable representation of the mapping or mappings.

Evolving and extending the DSVL

We hope to allow migration from an old repository to a new one using the C++ generated file. Currently, the repository is very brittle, but because we don't currently have anything to put in it, except for some nice demos, that's fine for now. A class will break existing repositories if a new persistent attribute is added. We might put the repository in a relational database, allowing us to manipulate it more easily or write a mapping to convert the old mapping files. We stopped innovating once the repository format was being used and began innovating again once the format was not being used. To support old development, we created a tag in our code repository should anyone wish to return to the old format. We can also store old TWB and TE executables.

Our new development is as follows. The TWB/TE has been recently extended to support:

- XML and EDI/X12 text in a form display as an outline
- a shell (can execute a program, send text to the program through a pipe, and retrieve text from the same program through a pipe)
- a SQL statement runner, including binding input and output variables
- developing TCP/IP servers.

To extend the language, you either need to add a new desktop object or a new operation. To add a new desktop object, you need an MVC representation of the object, a small icon, a large icon, a new tools menu item, constructors, a class id and a default label, in addition, you may require other methods to support existing operations such as copy. To add a new operation, you need an MVC representation of the

operation, an icon, operation constructors, a class id, an undo class and a play method. These must be provided in C++ or C++ wrapped utilities (our SQL is implemented with Java).

What our DSVL isn't

Our DSVL isn't Object Oriented or Structured. It's one huge program, with a huge number of global objects. It doesn't really raise the abstraction level of programming, except by removing much of the syntax and replacing it with learning a set of GUI tools. My claim is that you can't abstract away all condition tests in any complex environment you want to run on a computer. And you don't want to abstract away things that provide modularity and reuse.

MetaWhat??????

I have several questions. If I want to generalize/abstract this environment, what do I do? For textual languages, you could start writing a parser-generator to support generation of new languages. What do I do for a visual programming language? What appears to me to be common across all visual programming and visual language environments is the concept of the metaphor or maybe Model-View-Controller (MVC). I think that the metamodelling and metaCASE tools do this very well. But how practical are they for building an imperative or declarative programming environment? Are there metaprogramming environments and metainterpreters out there for visual programming languages? Is CASE metaprogramming? Are grammars metaprogramming? How about creating new IDEs for cross language development, including visual programming? If you have metaCASE, do you need another programming environment to run your programs in? How are imperative features supported within metaCASE? What we use is the flowchart with domain specific operations embedded within it.

Conclusion

Thank you for reading this document. I appreciate your patience if this is all old news to you. We actually started developing this in 1991, and it hasn't been published officially since them. I would like to thank Jeff Allen and Frank Guerra who made the environment viable as well as Ted Cole, who served as acting project leader and domain expert. I hope I've convinced my readers that domain-specific visual programming languages are possible and practical. Maybe we'll see more domain-specific visual programming languages, and visual programming language construction environments in the future.